Recall we defined running times in terms of **basic computer steps.**

Basic computer steps include all arithmetic operations with **small numbers**

Small numbers : Can be represented by a constant number of bits in memory. Recall that a number b requires $O(\log_2 b)$ bits.

Generally in this class, we will work only with small numbers, but we'll take a small detour and study algorithms for doing arithmetic with large numbers.

Sumation Algorithm

Input Two n bit integers X and Y (in binary)

Output $X+Y$ (in binary)

Example

$$\begin{array}{rrrrrrrrr} X = & 1 & 1 & 0 & 1 & 0 & 1 & & (53) \\ Y = & + 1 & 0 & 0 & 0 & 1 & 1 & & (35) \\ \hline X+Y = & 1 & 0 & 1 & 1 & 0 & 0 & 0 & (88) \end{array}$$

Correctness This is the algorithm for adding numbers from elementary school, so we know it is correct.

Running Time Can be implemented in $O(n)$ time with a loop going from right to left.

Can we do better? No. Reading the numbers takes $\Omega(n)$ time.

Note: Adding two n bit numbers may result in an $n+1$ bit number.

Multiplication Algorithm

Input Two n bit integers X and Y (in binary)

Output $X \cdot Y$ (in binary)

Example $X = 1101 \quad Y = 1011$

$$X \cdot Y = 1101 \cdot 1011$$

$$\begin{array}{r} 1101 \\ 0000 \\ 1101 \\ +\quad 1101 \\ \hline 10001111 \end{array}$$

Correctness This is the algorithm for multiplying numbers from elementary school, so we know it is correct

Running Time We are doing $O(n)$ sums between n bit numbers. Hence, each sum takes $O(n)$ time, and the running time is $O(n^2)$.

Can we do better? Yes! Soon we will study a multiplication algorithm with $O(n^{1.59})$ running time.

Historical Curiosity: The fastest known multiplication algorithm has $O(n \log n)$ running time. It was discovered in 2019!

The fastest known algorithm from 2007 to 2019 was discovered here at Penn State by Prof. Fürer.

Division

⇒ We've all learned different algorithms for division, but implementing the typical ones could be tricky.

⇒ We'll discuss an algorithm for division that is efficient and easy to implement.

Input: Two n-bit numbers X and Y

Output: Two integers q and r such that $X = q \cdot Y + r$.

r: remainder; $0 \le r \le Y-1$

q: quotient

⇒ The algorithm is recursive, and is based on the following ideas.

⇒ Suppose X is even and that $\frac{X}{2} = \hat{q} \cdot y + \hat{r}$ with $0 \le \hat{r} \le y-1$

$\Rightarrow$ Then, $x = 2\hat{q} \cdot y + 2\hat{r}$

$\Rightarrow$ So, if $(2\hat{r}) \leq y-1$, then $x = q \cdot y + r$ with $q = 2\hat{q}$ and $r = 2\hat{r}$

else $x = q \cdot y + r$ with $r = 2\hat{r} - y$ $q = 2\hat{q}+1$.

$\Rightarrow$ Like wise, if x is odd, and we know $\hat{q}$ and $\hat{r}$ such that:

$$\frac{x-1}{2} = \hat{q} \cdot y + \hat{r}$$

$\Rightarrow$ Then, $x = 2\hat{q}y + 2\hat{r}+1$

$\Rightarrow$ If $2\hat{r}+1 \leq y-1$, $x = qy+r$ with $q = 2\hat{q}$ and $r = 2\hat{r}+1$

else $x = qy+r$ with $q = 2\hat{q}+1$ $r = 2\hat{r}+1-y$.

Division (x, y)

Input Two n-bit numbers x and y.

Output (q, r) such that $x = q \cdot y + r$.

If $x == 0$ return $(0,0)$

$(\hat{q}, \hat{r}) =$ Division $\left(\lfloor \frac{x}{2} \rfloor, y\right)$

// Note that $\lfloor \frac{x}{2} \rfloor = \begin{cases} \frac{x}{2} & \text{if } x \text{ even} \\ \frac{x-1}{2} & \text{if } x \text{ odd} \end{cases}$

Set $q = 2\hat{q}$ $r = 2\hat{r}$

If x is odd: $r = r+1$.

If $r \geq y$: , $r = r-y$ $q = q+1$

return (q, r)

Correctness By induction. Base case $x=0$. In this case, the algorithm is correct.

Then assume that algorithm is correct on input $(\lfloor \frac{x}{2} \rfloor, y)$.

The discussion before the algorithm implies that it is correct for (x, y) and we are done.

Running Time:

- We have $O(\log x) = O(n)$ recursive calls.
- In each call we potentially do 2 multiplications by 2, and a parity check. Each take $O(1)$.
- The substraction, addition, and comparison take $O(n)$.
- Overall running time $O(n^2)$.

Can we do better?

- There are sophisticated method to reduce division of n-bit numbers to $O(1)$ multiplications of n-bit numbers.
- These results in $O(n \log n)$ algorithms.

Modular Exponentiation

Input Three n-bit integers X, Y and N

Output $X^Y \bmod N$

Beyond Naive Algorithm:

① Compute $L = \overbrace{X \cdot X \cdot X \cdots X}^{Y \text{ times}}$

② Compute $L \bmod N$

⊛ Requires $Y = 2^n$ multiplications!

⊛ Multiplications keep getting bigger and bigger!

$X^{Y/2}$ has $O(\log_2 X^{Y/2}) = O(2^n \cdot n)$ bits!!

Fast Exponentiation Algorithm

Two ideas:

① Reduce number size by doing all arithmetic mod N.

② Try to reduce number of multiplications.

Fact 1: $a \cdot b \bmod N = (a \bmod N) \cdot (b \bmod N)$

Proof: $a = k_a N + r_a$, $b = k_b N + r_b \Rightarrow (a \bmod N)(b \bmod N) = r_a \cdot r_b$

$$a \cdot b = (k_a N + r_a)(k_b N + r_b) = k_a \cdot k_b \cdot N^2 + (k_a \cdot r_b + k_b \cdot r_a) N + r_a \cdot r_b$$

$\Rightarrow a \cdot b \bmod N = r_a \cdot r_b$. ∎

⊛ With fact 1 in hand, we can do all the arithmetic mod N!!

Fact 2

- If Y is even, then $X^Y = X^{Y/2} \cdot X^{Y/2}$
- If Y is odd, then $X^Y = X^{(Y-1)/2} \cdot X^{(Y-1)/2} \cdot X$

Fast-Exponentiation (X, Y, N)

Input: Three n-bit numbers X, Y and N

Output: $X^Y \bmod N$

```
If Y == 0   return 1
If Y is even
    z = Fast-Exponentiation(X, Y/2, N)
    return z·z mod N
If Y is odd
    z = Fast-Exponentiation(X, (Y-1)/2, N)
    return z·z·X mod N
```

Correctness Follows from Facts 1 and 2 above

Running Time:

- The number of recursive calls is $O(\log_2 Y) = O(n)$
- In each call, we do $O(1)$ multiplications and one division with n-bit numbers; this takes $O(n \log n)$
- Overall $O(n^2 \log n)$; **Big Improvement!!**

Algorithms for Fibonacci Sequence

Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n \geq 2 \end{cases}$$

This sequence was introduced in 1200's to calculate growth of rabbit populations.

Input: integer n

Output: n-th Fibonacci number.

Fact. $\forall n \geq 9 \quad 2^{0.2n} \geq F_n \geq 2^{0.5n}$

Proof. Exercise. (Check Problem 0.3 from textbook)

First try:

```
Fib1(n)
    If n==0  return 0;
    If n==1  return 1;
    return Fib1(n-1) + Fib1(n-2);
```

Correctness Follows from the definition of F_n.

Running Time $T(n)$ = # of basic computer steps needed to compute F_n. Then:

$T(n) \geq T(n-1) + T(n-2)$, $T(1) \geq 1$ and $T(0) \geq 1$.

Therefore $T(n) \geq F_n \geq 2^{0.5n}$

Note: Computing F_{200} will require at least 2^{100} steps. In the fastest computers in the world this will require $\geq 300\,000$ years.

Can we do better? Yes.

```
Fib2(n)
    if n == 0   return 0;
    if n == 1   return 1;
    int[] f = new int[n];
    f[0] = 0
    f[1] = 1
    for i=2 to n
        f[i] = f[i-1] + f[i-2]
    return f[n];
```

Correctness Follows from the definition of F_n.

Running Time

- $O(n)$ steps. (mostly additions)
- But F_n is an n bit number, so the addition inside the loop takes $O(n)$.
- So, overall we get $O(n^2)$ running time.

Now we can compute F_{100000} in seconds!

Can we do better? We can in fact do better. There is a more sophisticated algorithm with $O(n(\log n)^2)$ running time. (Check Problem 0.4 in your textbook).